Chapter 14

Formal Methods for Verifications of Reactive Systems

Olfa Mosbahi
Nancy University, France & Martin Luther University, Germany

Mohamed Khalgui
Xidian University, China

ABSTRACT

This chapter deals with the use of two verification approaches: theorem proving and model checking. The authors focus on the Event-B method by using its associated theorem proving tool (Click_n_Prove), and on the language TLA+ by using its model checker TLC. By considering the limitation of the Event-B method to invariance properties, the authors propose to apply the language TLA+ to verify liveness properties on a software behavior. The authors extend first the expressivity and the semantics of a B model (called temporal B model) to deal with the specification of fairness and eventuality properties. Second, they give transformation rules from a temporal B model into a TLA+ module. The authors present in particular, their prototype system called B2TLA+, that they have developed to support this transformation; then they can verify these properties thanks to the model checker TLC on finite state systems. For the verification of infinite-state systems, they propose the use of the predicate diagrams. The authors illustrate their approach on a case study of a parcel sorting system.

INTRODUCTION

Reactive systems are systems whose role is to maintain an ongoing interaction with their environment rather than produce some final values upon termination. Typical examples of reactive systems are Air traffic control systems, Programs controlling mechanical devices such as a train, a plane, or ongoing processes such as a nuclear reactor. Formal methods [Spivey, 1988] is the term used to describe the specification and verification
of these systems using mathematical and logical techniques. The main advantages of the formal approach to software construction [Dyba, Kampenes & Sjøberg, 2006; Pickard, Kitchenham & Jones, 1998; Wohlin, 2005; Wordsworth, 1987] is that, whenever applicable, it can lead to an increase of the reliability and correctness of the resulting programs by several orders of magnitude.

Several approaches for the verification of reactive systems are available, the most prominent are Algorithmic (model checking [Clarke, Grumberg, Jha, Lu & Veith, 2001; Clarke, Emerson & Sistla, 1986; Clarke, 1994; Clarke, 1997; Clarke, Grumberg & Peled, 1999; Kaltenbach, 1994]) and Deductive verification (theorem-proving techniques [Hoare, 1969; Dijkstra, 1975; Kaufmann & Moore, 2004; Archer, Vito & Munro, 2003; Holzmann, 2003]). These approaches are used to establish the correctness of reactive programs relative to their temporal specifications. Verifying the correctness of a program involves formulating a property to be verified using a suitable logic such as first order logic or temporal logic.

The model checking problem [Clarke, Grumberg, Jha, Lu & Veith, 2001; Clarke, Emerson & Sistla, 1986] involves the construction of an abstract model $M$, in the form of variations on finite state automata, and the construction of specification formulas $\varphi$, in the form of variations on temporal logic. The verification algorithm used in model checking involves exploring the set of reachable states of the model to ensure that the formula $\varphi$ holds. It has gained popularity in industry because the verification procedure can be fully automated [Clarke, Emerson & Sistla, 1986] and counterexamples are automatically generated if the property being verified does not hold. Furthermore, model checkers rely on exhaustive state space enumeration to establish whether a property holds or doesn’t hold. This approach to verification puts immediate limits on the state space of problems that can be explored by model checkers. This common problem, known as the state explosion problem, is an often cited drawback of verification by model checking.

Theorem proving is a very tedious process involving keeping in mind a multitude of assumptions and transformation rules. The calculi used in theorem proving is based on Hoare and Dijkstra theories [Dijkstra, 1975]. The first one describes a calculus to reason about program correctness in terms of pre and post conditions [Dijkstra, 1976; Dijkstra & Schweten, 1990]. Dijkstra extended Hoare’s ideas in the concept of “predicate transformers” which, instead of starting with a pre-condition and post-condition, starts with a post-condition and uses the program code to determine the pre-condition that needs to hold to establish the post condition [Dijkstra, 1975; Dijkstra, 1995]. Hoare’s approach to proving correctness introduced the concept of a “Hoare triple”, which is a formula in the form: $\{\varphi_{\text{pre}}\} P \{\varphi_{\text{post}}\}$. This formula can be read as “if property $\varphi_{\text{pre}}$ holds before program $P$ starts, $\varphi_{\text{post}}$ holds after the execution of $P$”. The program $P$ can refer to an entire program or to a single function call, depending on the unit that is being verified. In Hoare’s calculus, axioms and rules of inference are used to derive $\{\varphi_{\text{post}}\}$ based on $\{\varphi_{\text{pre}}\}$ and $P$. The syntax of $P$ described by Hoare corresponds to a simple imperative language with the usual constructs (assignment, conditional branching, looping, and sequential statements).

A key difference between the theorem approach to software verification and the model checking approach to software verification is that theorem provers do not need to exhaustively visit the program’s state space to verify properties. Consequently, a theorem prover approach can reason about infinite state spaces and state spaces involving complex datatypes and recursion. This can be achieved because a theorem prover reasons about constraints on states, not instances of states. While theorem provers have distinct advantages over model checkers, namely in the superior size of the systems they can be applied to and their ability to reason inductively, deductive systems